Arrays

Variables, as you might remember, are a specific section of memory that holds a specific type of value. We could visualize two int variables like this:

int foo = 13; **foo bar**

**13**

**50**

int bar = 50;

An **array** is a group of variables (all of the same type) that are placed sequentially in memory. Arrays are treated very similarly in C and Java, but they are declared differently. The example below shows how to declare an array of ten integers.

|  |  |
| --- | --- |
| **C** | **Java** |
| int myList[10]; | int myList[] = new int[10]; |

We can think of this array visually like this:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

**myList**

Each of these boxes (called **elements**) can hold a different int value, but we only have one variable name, myList. To access each of the ten values we use an **index** inside of square brackets, like myList[4]. The square brackets are called the **subscript** operator.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

**myList**

myList[0] myList[1] myList[2] myList[3] myList[4] myList[5] myList[6] myList[7] myList[8] myList[9]

Array indexes in C and Java start at 0. This is very important to remember. Note that this means that if you want to access the sixth element of the array you need to use an index of 5.

In Java, if you use an index that is outside the array (like trying to do myList[10] with the above example), your application will crash with an “Array Index Out of Bounds” exception. In C, however, nothing will prevent you from reading from or writing to indexes that are out of bounds. Your program may even appear to work correctly. Most of the time. This can be a source of random-seeming and tricky bugs, so always double-check your indexes to make sure you are not going past the end of your arrays.

# Initialization

In both C and Java you can set the array values when you initialize it. The array will be created with the number of elements that you put inside the curly brackets, and each element will be set to its respective value.

int myList[] = { 2, 6, 5, 7, 1};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2** | **6** | **5** | **7** | **1** |

**myList**

# Iteration

Whenever we need to do something with every element of an array, we can easily iterate through it using a for loop.

|  |  |
| --- | --- |
| **C** | **Java** |
| int i;  int array[10];  for (i = 0; i < 10; ++i)  {  array[i] = i \* 2;  } | int array[] = new int[10];  for (int i = 0; i < 10; ++i)  {  array[i] = i \* 2;  } |
| int i;  int array[5] = { 2, 5, 7, 3, 2 };  for (i = 0; i < 5; ++i)  {  printf("%d\n", array[i]);  } | int array[] = { 2, 5, 7, 3, 2 };  for (int i = 0; i < 5; ++i)  {  System.out.println(array[i]);  } |

Remember that array indexes start with zero, which means that the last index will be one less than the array’s length, so we have to make sure to use the < operator in our check for terminating the loop, not the <= operator.

# Assignment

Unlike other variables, we can’t use the assignment operator to make one array the same as another, like array1 = array2. Instead we have to manually copy each element from one array to the other. Naturally, both arrays must be the same size for this to work correctly.

|  |  |
| --- | --- |
| **C** | **Java** |
| int i;  int array1[5] = { 2, 5, 7, 3, 2 };  int array2[5];  for (i = 0; i < 10; ++i)  {  array2[i] = array1[i];  } | int array1[] = { 2, 5, 7, 3, 2 };  int array2[] = new int[5];  for (int i = 0; i < 10; ++i)  {  array2[i] = array1[i];  } |

# Array Length in Java

In Java we have an easy way to find out from the array itself what the length of the array is. One of the ways Java arrays are different from C arrays is that they are an object with properties. The **length** property is a variable inside the array object which stores the number of elements in that array.

The dot operator is used to access an object’s properties:

int array[] = new int[10];

int size = array.length;

This means that when using for loops, you don’t have to specify the exact number of iterations, so your code can be more generic and flexible.

int array[] = { 2, 5, 7, 3, 2 };

for (int i = 0; i < **array.length**; ++i)

{

System.out.println(array[i]);

}

# Size Define in C

Although we don’t have array properties in C, it is still a good idea to not use specific numbers for the size of our arrays in for loops and other areas. If the size of the array changes, and we forget to update a loop, we could end up either not updating enough of the array elements or running off the end of the array and causing bugs.

An easy way around this is by using a **#define** for the array size. We give the define a name and a value, and then the compiler will simply substitute the value for the name when it compiles the code. Since this is still a constant value we can create our arrays using the defined name, as well as using it in the rest of the code referencing that array. Then, if we need to change the size of the array, we only have to change it in one place.

Note that there is no semicolon at the end of the define. If you included one, the example below would replace SIZE with “10;” instead of “10”, and your code would not compile. It is a common convention to make the names of defines in all capital letters so they don’t look like variables.

#define SIZE 10

int array[SIZE];

int i;

for (i = 0; i < SIZE; ++i)

{

array[i] = i;

}